Proyecto Final

Nombre del catedrático: Marco Waldo Ángeles Tena

Materia: Programación Concurrente

Nivel: Superior

Carrera: Ingeniería en Software

Nombre de los Alumnos: Fabricio Meneses Ávila, Jorge Ruiz Diaz, Josefa Francisco Hernández, Diego Daniel Magdaleno Medina, Ángel Gabriel Castillo Sánchez

Matriculas: 2231122171, 2231122197, 2231122164 2231122172, 2231122204

Fecha de elaboración : 20/Noviembre/2024

Grupo: SW_07_03

Indice

Objetivo General

Crear un programa con GUI que facilite la ejecución de scripts prácticos enfocados en hilos, semáforos, patrones de diseño y comunicación entre procesos, usando Python y Tkinter.

Objetivos Específicos

- Diseñar una GUI intuitiva para ejecutar scripts relacionados con programación concurrente.
- Desarrollar módulos para demostrar manejo de hilos, sincronización y comunicación entre procesos.
- Incluir un visor PDF para documentación integrada.
- Ofrecer herramientas educativas para practicar conceptos avanzados.
- Asegurar un diseño modular y escalable para futuras extensiones.

Descripción del Programa

Secciones principales:

Menú Principal y Submenús:

- Hilos: Ejecución concurrente con sincronización.
- Sockets: Comunicación cliente-servidor con TCP/UDP.
- Semáforos: Problemas clásicos (Condición de Carrera, Barbero Dormilón).
- Patrones: Implementación de patrones como Futuro/Promesa y Reactor/Proactor.
- Documentación: Visor de PDF integrado con los apuntes y documentación del programa.
- Ayuda: Información sobre los desarrolladores.
- Salir: Cierra el programa.

Cada sección tiene submenús con botones para ejecutar scripts específicos.

Importaciones:

- Tkinter: Usado para crear la interfaz gráfica de usuario.
- PIL (Pillow): Usado para manipular y mostrar imágenes, en este caso, para mostrar documentos PDF en forma de imágenes.
- fitz: Librería de PyMuPDF que se utiliza para abrir y manipular archivos PDF.
- os: Proporciona funciones para interactuar con el sistema operativo, como acceder a rutas de archivos.
- Módulos importados (Hilos, Sockets, Semáforos, Patrones): Estos módulos contienen diversas funciones y ejemplos relacionados con la programación concurrente.

Variables Globales:

ventanas_abiertas: Lista que mantiene un registro de las ventanas abiertas en la aplicación, lo cual es útil para cerrarlas correctamente cuando se cierra la aplicación principal.

PDF_PATHS: Diccionario que mapea nombres de documentos de documentación a sus rutas de archivo correspondientes. Estos documentos son PDFs que explican diversos conceptos de programación concurrente.

Funciones Principales:

- mostrar_pdf(pdf_path): Esta función abre una ventana emergente (Toplevel) que carga un archivo PDF especificado en pdf_path y lo muestra en la interfaz como una serie de imágenes (cada página del PDF se convierte en una imagen y se muestra en un Label de Tkinter).

- opciones_funciones: Diccionario que mapea las opciones del menú a funciones específicas importadas desde los módulos relacionados con hilos, sockets, semáforos, patrones de concurrencia, etc. Si la función no está disponible en un módulo, se asigna una función de reemplazo que simplemente imprime un mensaje de error.

- mostrar_submenu(titulo, opciones): Esta función actualiza la interfaz con un submenú que muestra opciones basadas en el título y las funciones asociadas. Por ejemplo, si el título es "Hilos", se mostrarán las opciones relacionadas con hilos, y al seleccionar una opción, se ejecutará la función correspondiente.

- cerrar_aplicacion(): Cierra la aplicación y todas las ventanas emergentes que se hayan abierto. Llama a os._exit(0) para asegurar que el proceso se termine correctamente.

- mostrar_acerca_de(): Muestra una ventana emergente que contiene información sobre el proyecto, como los nombres y matrículas de los integrantes del equipo de desarrollo.

Interfaz Gráfica (GUI):

- root: La ventana principal de la aplicación, que es de tipo Tk (objeto principal de Tkinter).
- Imagen de fondo (bg_image): Carga una imagen que se usa como fondo de la ventana principal.
- Menú principal (menu_bar): Un conjunto de botones en la parte superior de la ventana que actúan como menú. Cada botón abre un submenú con opciones relacionadas con hilos, sockets, semáforos, patrones de diseño, y documentación. Al seleccionar una opción, se ejecuta la función correspondiente.

Opciones del Menú:

Hilos, Sockets, Semáforos, Patrones: Al hacer clic en uno de estos botones del menú, se muestran opciones relacionadas con cada tema. Estas opciones están asociadas con funciones que implementan ejemplos prácticos o teorías de concurrencia (como el patrón productor-consumidor, semáforos, etc.).

Documentación: Muestra un submenú con una lista de documentos PDF que explican diversos temas de programación concurrente.

Acerca de: Muestra una ventana con la información sobre el proyecto y los integrantes.

Salir: Cierra la aplicación.

Estilo y Diseño:

Se usan colores y fuentes para hacer la interfaz más atractiva y fácil de usar. Los botones tienen un estilo consistente con colores azules y texto en blanco.

Se utiliza un fondo de color gris oscuro y texto en blanco para la mayoría de los elementos, lo que hace que la aplicación sea visualmente clara y legible.

Ejecución:

root.mainloop(): Este es el bucle principal de Tkinter que mantiene la aplicación abierta, esperando la interacción del usuario (hacer clic en los botones, cerrar ventanas, etc.).

Descripción y códigos de programas de menú

Definiciones:

Hilos:

Un hilo (thread) es una unidad de ejecución dentro de un proceso. Los hilos comparten el mismo espacio de memoria y recursos del proceso principal, pero se ejecutan de manera concurrente, lo que permite que un programa realice múltiples tareas simultáneamente. Esto es útil en aplicaciones que requieren procesamiento paralelo o en las que se deben realizar operaciones sin bloquear la interfaz de usuario.

```python
import threading
import tkinter as tk
from tkinter import messagebox

# Función que se ejecutará en el hilo
def Primer_Hilo(label):
    # Actualizar el texto en el widget de la etiqueta
    label.config(text="Mi Primer Programa con Hilos en Python")

# Función principal para ejecutar este módulo
def ejecutar():
    # Crear la ventana principal de tkinter
    root = tk.Tk()
    root.title("Hilos-Hilos")
    root.geometry("400x150")
    root.configure(bg="#34495e")  # Fondo gris oscuro

    # Crear y empaquetar una etiqueta en la ventana
    label = tk.Label(root, text="Esperando ejecución del hilo...", fg="#ffffff", bg="#34495e")
    label.pack(pady=20)

    # Función para iniciar el hilo
    def iniciar_hilo():
        thread = threading.Thread(target=Primer_Hilo, args=(label,))
        thread.start()

    # Crear y empaquetar un botón en la ventana
    boton = tk.Button(root, text="Iniciar Hilo", command=iniciar_hilo, bg="#3498db", fg="#ffffff", activebackground="#2980b9", activeforeground="#ffffff")
    boton.pack(pady=10)

    # Iniciar el bucle principal de tkinter
    root.mainloop()

# Ejecutar la función principal
if __name__ == "__main__":
    ejecutar()

```

Sockets:

Un socket es una interfaz para la comunicación entre programas en una red. Proporciona un punto final para enviar y recibir datos a través de un protocolo de red, como TCP/IP. Los sockets permiten que los procesos se comuniquen entre sí, incluso si están ejecutándose en máquinas diferentes

```python
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext
from queue import Queue

# Dirección y puerto
direccion = "localhost"
puerto = 9999

# Cola para mensajes del cliente y servidor
server_queue = Queue()
client_queue = Queue()

def ejecutar():
    def iniciar_servidor():
        def servidor():
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as mySocket:
                try:
                    mySocket.bind((direccion, puerto))
                    server_queue.put(f"Servidor iniciado en {direccion}:{puerto}\n")
                    mySocket.listen(5)
                    server_queue.put("Esperando conexiones...\n")

                    client, client_addr = mySocket.accept()
                    server_queue.put(f"Conexión establecida con {client_addr}\n")

                    msg = client.recv(1024).decode()
                    server_queue.put(f"Mensaje recibido del cliente: {msg}\n")

                    msg_out = f"Mensaje recibido: {msg}. Gracias."
                    client.send(msg_out.encode())
                    server_queue.put(f"Enviando acuse de recibo: {msg_out}\n")

                    client.close()
                    server_queue.put("Conexión cerrada con el cliente.\n")
                except OSError as e:
                    if e.errno == 10048:
                        server_queue.put(f"Error: El puerto {puerto} ya está en uso.\n")
                    else:
                        server_queue.put(f"Error en el servidor: {e}\n")
                except Exception as e:
                    server_queue.put(f"Error en el servidor: {e}\n")

        threading.Thread(target=servidor, daemon=True).start()

    def iniciar_cliente():
        def cliente():
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as mySocket:
```

```
try:
                    mySocket.connect((direccion, puerto))
                    client_queue.put(f"Conexión establecida con el servidor en {direccion}:{puerto}\n")

                    msg = "Hola, soy un cliente TCP creado por Equipo 4."
                    client_queue.put(f"Enviando mensaje al servidor: {msg}\n")
                    mySocket.send(msg.encode())

                    msg_in = mySocket.recv(1024).decode()
                    client_queue.put(f"Acuse de recibo del servidor: {msg_in}\n")

                    mySocket.close()
                    client_queue.put("Conexión cerrada con el servidor.\n")
                except Exception as e:
                    client_queue.put(f"Error en el cliente: {e}\n")

        threading.Thread(target=cliente, daemon=True).start()

    def procesar_server_queue():
        while not server_queue.empty():
            msg = server_queue.get()
            server_output.insert(tk.END, msg)
            server_output.see(tk.END)
        server_root.after(100, procesar_server_queue)

    def procesar_client_queue():
        while not client_queue.empty():
            msg = client_queue.get()
            client_output.insert(tk.END, msg)
            client_output.see(tk.END)
        client_root.after(100, procesar_client_queue)

    # Ventana para el servidor
    server_root = tk.Tk()
    server_root.title("Servidor TCP")
    server_root.configure(bg="#34495e")

    server_output = scrolledtext.ScrolledText(
        server_root, wrap=tk.WORD, width=60, height=20, bg="#ffffff", fg="#000000"
    )
    server_output.pack(pady=20)

    server_button = tk.Button(
        server_root,
        text="Iniciar Servidor",
        bg="#3498db",
        fg="#ffffff",
        activebackground="#2c3e50",
        activeforeground="#ffffff",
        relief="flat",
        padx=10,
        pady=5,
    )
    server_button.configure(command=iniciar_servidor)
    server_button.pack(pady=10)

    # Ventana para el cliente
    client_root = tk.Tk()
    client_root.title("Cliente TCP")
    client_root.configure(bg="#34495e")

    client_output = scrolledtext.ScrolledText(
        client_root, wrap=tk.WORD, width=60, height=20, bg="#ffffff", fg="#000000"
    )
    client_output.pack(pady=20)

    client_button = tk.Button(
        client_root,
        text="Iniciar Cliente",
        bg="#3498db",
        fg="#ffffff",
        activebackground="#2c3e50",
        activeforeground="#ffffff",
        relief="flat",
        padx=10,
        pady=5,
    )
    client_button.configure(command=iniciar_cliente)
    client_button.pack(pady=10)

    # Procesar colas periódicamente
    server_root.after(100, procesar_server_queue)
    client_root.after(100, procesar_client_queue)

    # Ejecutar ambas ventanas
    threading.Thread(target=server_root.mainloop).start()
    client_root.mainloop()

```

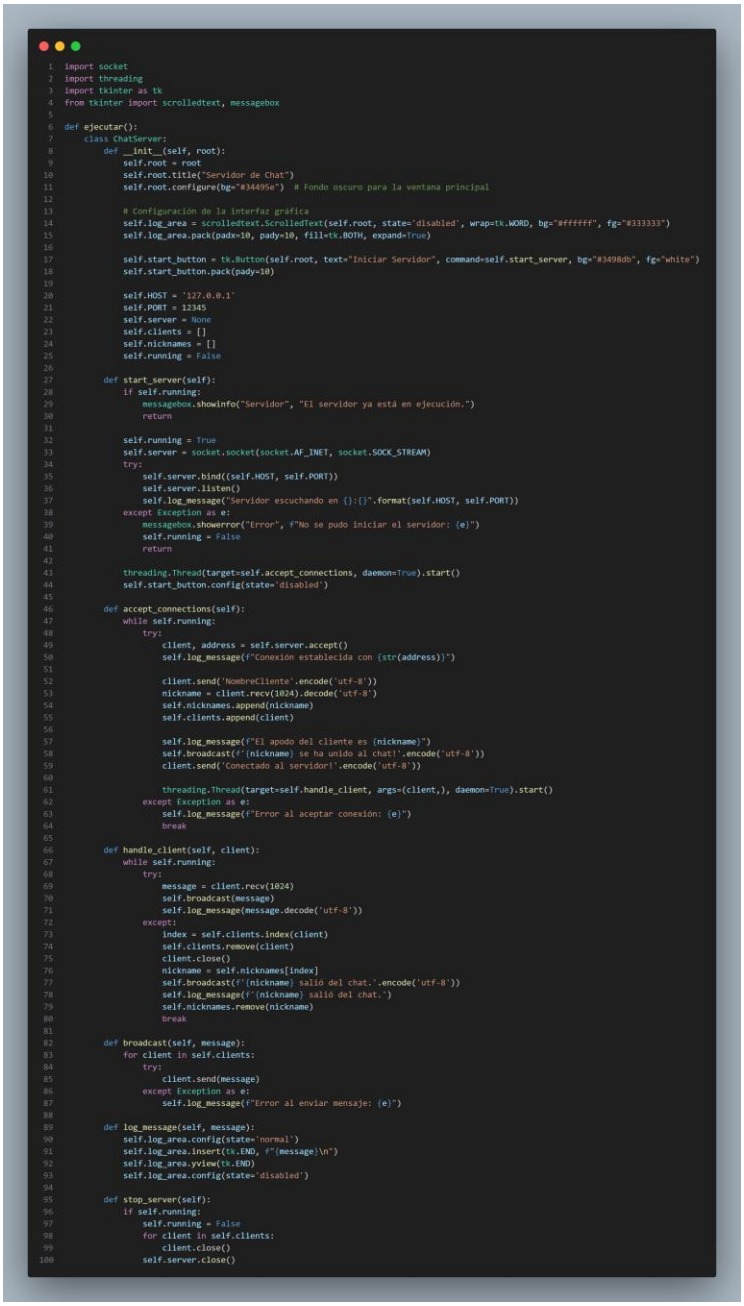
Semáforos:

Un semáforo es una variable o tipo de sincronización utilizada para controlar el acceso a un recurso compartido en un entorno de ejecución concurrente. Se utiliza para evitar condiciones de carrera y garantizar que los procesos accedan de manera segura a recursos compartidos, como memoria o dispositivos. Los semáforos se pueden utilizar para gestionar el número de procesos que pueden acceder a un recurso simultáneamente.

```python
import socket
import threading
from threading import Semaphore
import tkinter as tk

# Configuración del servidor
HOST = '127.0.0.1'
PORT = 65432
MAX_CLIENTES_SIMULTANEOS = 3

semaforo = Semaphore(MAX_CLIENTES_SIMULTANEOS)

def actualizar_mensaje(mensaje, text_widget):
    text_widget.insert(tk.END, mensaje + "\n")
    text_widget.yview(tk.END)

def manejar_cliente(conexion, direccion, text_widget):
    with semaforo:
        mensaje = conexion.recv(1024).decode('utf-8')
        respuesta = "Mensaje recibido y confirmado"

        actualizar_mensaje(f"Cliente {direccion}: {mensaje}", text_widget)
        conexion.sendall(respuesta.encode('utf-8'))
        actualizar_mensaje(f"Servidor: {respuesta}", text_widget)

        conexion.close()

def iniciar_servidor(text_widget):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as servidor:
        servidor.bind((HOST, PORT))
        servidor.listen(5)
        actualizar_mensaje(f"Servidor escuchando en {HOST}:{PORT}", text_widget)

        while True:
            conexion, direccion = servidor.accept()
            hilo_cliente = threading.Thread(target=manejar_cliente, args=(conexion, direccion, text_widget))
            hilo_cliente.start()

def iniciar_cliente(text_widget):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cliente:
        cliente.connect((HOST, PORT))
        mensaje = "Hola, servidor"
        cliente.sendall(mensaje.encode('utf-8'))

        respuesta = cliente.recv(1024).decode('utf-8')

        actualizar_mensaje(f"Cliente: {mensaje}", text_widget)
        actualizar_mensaje(f"Servidor: {respuesta}", text_widget)

def ejecutar():
    def iniciar_servidor_en_hilo(text_widget):
        servidor_thread = threading.Thread(target=iniciar_servidor, args=(text_widget,))
        servidor_thread.daemon = True
        servidor_thread.start()

    def iniciar_cliente_en_hilo(text_widget):
        cliente_thread = threading.Thread(target=iniciar_cliente, args=(text_widget,))
        cliente_thread.daemon = True
        cliente_thread.start()

    def crear_ventana_servidor():
        ventana_servidor = tk.Tk()
        ventana_servidor.title("Servidor")
        ventana_servidor.configure(bg='lightblue')

        text_widget_servidor = tk.Text(ventana_servidor, height=20, width=80, wrap=tk.WORD, bg='white', fg='black')
        text_widget_servidor.pack(padx=10, pady=10)

        tk.Button(ventana_servidor, text="Iniciar Servidor", command=lambda: iniciar_servidor_en_hilo(text_widget_servidor), width=20, height=2, bg='green', fg='white').pack(pady=5)

        ventana_servidor.mainloop()

    def crear_ventana_cliente():
        ventana_cliente = tk.Tk()
        ventana_cliente.title("Cliente")
        ventana_cliente.configure(bg='lightgreen')

        text_widget_cliente = tk.Text(ventana_cliente, height=20, width=80, wrap=tk.WORD, bg='white', fg='black')
        text_widget_cliente.pack(padx=10, pady=10)

        tk.Button(ventana_cliente, text="Iniciar Cliente", command=lambda: iniciar_cliente_en_hilo(text_widget_cliente), width=20, height=2, bg='blue', fg='white').pack(pady=5)

        ventana_cliente.mainloop()

    hilo_ventana_servidor = threading.Thread(target=crear_ventana_servidor)
    hilo_ventana_servidor.daemon = True
    hilo_ventana_servidor.start()

    hilo_ventana_cliente = threading.Thread(target=crear_ventana_cliente)
    hilo_ventana_cliente.daemon = True
    hilo_ventana_cliente.start()

    tk.mainloop()
```

Sala de Chat:

Una sala de chat es un entorno virtual donde varios usuarios pueden intercambiar mensajes de texto en tiempo real. Generalmente, es una plataforma en línea donde las personas se comunican de manera instantánea, ya sea de manera privada o en grupos. En términos técnicos, puede implementarse utilizando redes de sockets para gestionar la comunicación entre los usuarios.



```python
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox

def ejecutar():
    class ChatServer:
        def __init__(self, root):
            self.root = root
            self.root.title("Servidor de Chat")
            self.root.configure(bg="#34495e")  # Fondo oscuro para la ventana principal

            # Configuración de la interfaz gráfica
            self.log_area = scrolledtext.ScrolledText(self.root, state='disabled', wrap=tk.WORD, bg="#ffffff", fg="#333333")
            self.log_area.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

            self.start_button = tk.Button(self.root, text="Iniciar Servidor", command=self.start_server, bg="#3498db", fg="white")
            self.start_button.pack(pady=10)

            self.HOST = '127.0.0.1'
            self.PORT = 12345
            self.server = None
            self.clients = []
            self.nicknames = []
            self.running = False

        def start_server(self):
            if self.running:
                messagebox.showinfo("Servidor", "El servidor ya está en ejecución.")
                return

            self.running = True
            self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            try:
                self.server.bind((self.HOST, self.PORT))
                self.server.listen()
                self.log_message("Servidor escuchando en {}:{}".format(self.HOST, self.PORT))
            except Exception as e:
                messagebox.showerror("Error", f"No se pudo iniciar el servidor: {e}")
                self.running = False
                return

            threading.Thread(target=self.accept_connections, daemon=True).start()
            self.start_button.config(state='disabled')

        def accept_connections(self):
            while self.running:
                try:
                    client, address = self.server.accept()
                    self.log_message(f"Conexión establecida con {str(address)}")

                    client.send('NombreCliente'.encode('utf-8'))
                    nickname = client.recv(1024).decode('utf-8')
                    self.nicknames.append(nickname)
                    self.clients.append(client)

                    self.log_message(f"El apodo del cliente es {nickname}")
                    self.broadcast(f'{nickname} se ha unido al chat!'.encode('utf-8'))
                    client.send('Conectado al servidor!'.encode('utf-8'))

                    threading.Thread(target=self.handle_client, args=(client,), daemon=True).start()
                except Exception as e:
                    self.log_message(f"Error al aceptar conexión: {e}")
                    break

        def handle_client(self, client):
            while self.running:
                try:
                    message = client.recv(1024)
                    self.broadcast(message)
                    self.log_message(message.decode('utf-8'))
                except:
                    index = self.clients.index(client)
                    self.clients.remove(client)
                    client.close()
                    nickname = self.nicknames[index]
                    self.broadcast(f'{nickname} salió del chat.'.encode('utf-8'))
                    self.log_message(f'{nickname} salió del chat.')
                    self.nicknames.remove(nickname)
                    break

        def broadcast(self, message):
            for client in self.clients:
                try:
                    client.send(message)
                except Exception as e:
                    self.log_message(f"Error al enviar mensaje: {e}")

        def log_message(self, message):
            self.log_area.config(state='normal')
            self.log_area.insert(tk.END, f"{message}\n")
            self.log_area.yview(tk.END)
            self.log_area.config(state='disabled')

        def stop_server(self):
            if self.running:
                self.running = False
                for client in self.clients:
                    client.close()
                self.server.close()
```

```
    self.log_message("Servidor detenido.")
            self.start_button.config(state='normal')

class ChatClient:
    def __init__(self, root):
        self.root = root
        self.root.title("Cliente de Chat")
        self.root.configure(bg="#34495e")  # Fondo oscuro para la ventana principal

        # Configuración del socket
        self.HOST = '127.0.0.1'
        self.PORT = 12345
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Configuración de la interfaz gráfica
        self.nickname_label = tk.Label(self.root, text="Introduce tu apodo:", bg="#34495e", fg="#ffffff")
        self.nickname_label.pack(padx=10, pady=5)

        self.nickname_entry = tk.Entry(self.root)
        self.nickname_entry.pack(padx=10, pady=5)
        self.nickname_entry.bind("<Return>", self.set_nickname)

        self.connect_button = tk.Button(self.root, text="Conectar", command=self.set_nickname, bg="#3498db", fg="white")
        self.connect_button.pack(padx=10, pady=5)

        self.chat_area = scrolledtext.ScrolledText(self.root, state='disabled', wrap=tk.WORD, bg="#ffffff", fg="#333333")
        self.input_field = tk.Entry(self.root)
        self.send_button = tk.Button(self.root, text="Enviar", command=self.write_message, bg="#3498db", fg="white")

    def set_nickname(self, event=None):
        self.nickname = self.nickname_entry.get().strip()
        if not self.nickname:
            messagebox.showwarning("Advertencia", "El apodo no puede estar vacío.")
            return

        self.nickname_label.pack_forget()
        self.nickname_entry.pack_forget()
        self.connect_button.pack_forget()

        self.chat_area.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)
        self.input_field.pack(padx=10, pady=10, fill=tk.X)
        self.input_field.bind("<Return>", self.write_message)
        self.send_button.pack(padx=10, pady=5)

        self.connect_to_server()

    def connect_to_server(self):
        try:
            self.client.connect((self.HOST, self.PORT))
        except Exception as e:
            messagebox.showerror("Error de conexión", f"No se pudo conectar al servidor: {e}")
            self.root.destroy()
            return

        threading.Thread(target=self.receive_messages, daemon=True).start()
        self.client.send(self.nickname.encode('ascii'))

    def receive_messages(self):
        while True:
            try:
                message = self.client.recv(1024).decode('ascii')
                if message == 'NombreCliente':
                    self.client.send(self.nickname.encode('ascii'))
                else:
                    self.update_chat_area(message)
            except Exception:
                self.update_chat_area("Ocurrió un error y se cerró la conexión.")
                self.client.close()
                break

    def write_message(self, event=None):
        message = self.input_field.get()
        if message.strip():
            full_message = f"{self.nickname}: {message}"
            self.client.send(full_message.encode('ascii'))
        self.input_field.delete(0, tk.END)

    def update_chat_area(self, message, from_self=False):
        self.chat_area.config(state='normal')
        self.chat_area.insert(tk.END, f"{message}\n", ("self_message" if from_self else ""))
        self.chat_area.tag_config("self_message", foreground="blue")
        self.chat_area.yview(tk.END)
        self.chat_area.config(state='disabled')

def main():
    server_thread = threading.Thread(target=start_server_window, daemon=True)
    client_thread1 = threading.Thread(target=start_client_window, daemon=True)
    client_thread2 = threading.Thread(target=start_client_window, daemon=True)
    client_thread3 = threading.Thread(target=start_client_window, daemon=True)

    server_thread.start()
    client_thread1.start()
    client_thread2.start()
    client_thread3.start()

    server_thread.join()
    client_thread1.join()
    client_thread2.join()
    client_thread3.join()

def start_server_window():
    server_root = tk.Tk()
    ChatServer(server_root)
    server_root.mainloop()

def start_client_window():
    client_root = tk.Tk()
    ChatClient(client_root)
    client_root.mainloop()

main()

if __name__ == "__main__":
    ejecutar()
```

Futuro – Promesa:

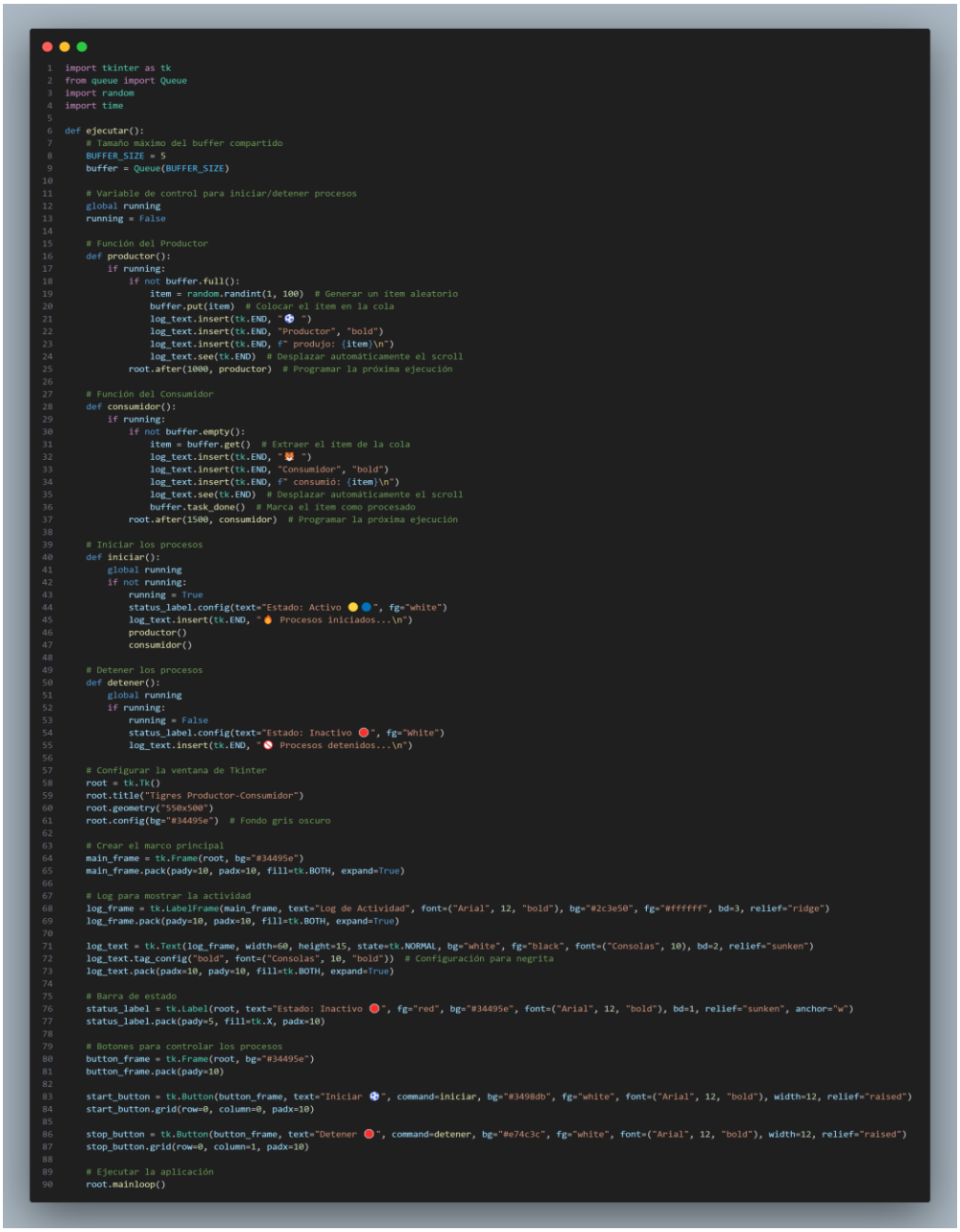
En programación, un futuro (future) o promesa (promise) es un mecanismo que representa un valor que estará disponible en algún momento en el futuro. Se utiliza en programación asincrónica para manejar operaciones que toman tiempo, como la lectura de archivos o solicitudes HTTP. El futuro permite que el programa continúe su ejecución mientras espera que se complete la operación, y luego maneja el resultado de manera asíncrona.

```python
import socket
import threading
import random
import time
import tkinter as tk
from tkinter import messagebox

# Configuración del servidor
HOST = '127.0.0.1'
PORT = 65432
MAX_CONNECTIONS = 5
def ejecutar():
    global server_running, server_socket
    # Semaphore para limitar las conexiones concurrentes
    semaphore = threading.Semaphore(MAX_CONNECTIONS)

    # Variable global para controlar el servidor
    server_running = False
    server_socket = None

    # Configuración de la ventana principal
    ventana_principal = tk.Tk()
    ventana_principal.title("Servidor Tigres")
    ventana_principal.geometry("600x500")
    ventana_principal.configure(bg="#34495e")  # Fondo gris oscuro

    # Título y frase alusiva
    etiqueta_titulo = tk.Label(
        ventana_principal,
        text="Servidor Tigres - ¡Unidos hasta el final!",
        font=("Arial", 16),
        bg="#34495e",  # Fondo gris oscuro
        fg="#ffffff"  # Texto blanco
    )
    etiqueta_titulo.pack(pady=10)

    # Agregar un cuadro de texto para mostrar el log
    cuadro_log = tk.Text(ventana_principal, height=20, width=70, state="disabled", font=("Arial", 10), bg="white", fg="black")
    cuadro_log.pack(pady=10)

    # Función para agregar mensajes al cuadro de log
    def agregar_log(mensaje):
        cuadro_log.config(state="normal")
        cuadro_log.insert(tk.END, f"{mensaje}\n")
        cuadro_log.config(state="disabled")
        cuadro_log.see(tk.END)

    # Función para manejar la conexión con un cliente
    def handle_client(connection, address):
        with connection:
            agregar_log(f"Cliente conectado desde {address}")
            sleep_time = random.uniform(1, 5)
            agregar_log(f"Procesando tarea para {address}, durará {sleep_time:.2f} segundos.")
            time.sleep(sleep_time)

            # Enviar respuesta al cliente
            message = f"Tarea completada en {sleep_time:.2f} segundos.\n"
            connection.sendall(message.encode('utf-8'))
            agregar_log(f"Tarea completada para {address}. Conexión cerrada.")

    # Función principal del servidor
    def server_task():
        global server_running, server_socket
        agregar_log(f"Iniciando servidor en {HOST}:{PORT}")
        server_running = True

```

```python

    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
            server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            server_socket.bind((HOST, PORT))
            server_socket.listen()
            agregar_log(f"Servidor escuchando en {HOST}:{PORT}")

            while server_running:
                try:
                    server_socket.settimeout(1.0)  # Tiempo de espera para aceptar conexiones
                    connection, address = server_socket.accept()
                    semaphore.acquire()
                    agregar_log(f"Conexión aceptada de {address}")

                    # Manejar el cliente en un hilo separado
                    thread = threading.Thread(target=handle_client, args=(connection, address))
                    thread.start()
                    thread.join()
                    semaphore.release()
                except socket.timeout:
                    continue  # Salir del bucle si no hay conexiones y server_running es False
    except Exception as e:
        agregar_log(f"Error del servidor: {e}")
    finally:
        agregar_log("Servidor detenido.")
        server_running = False

# Botón para iniciar el servidor
def iniciar_servidor():
    global server_running
    if (server_running):
        agregar_log("El servidor ya está en ejecución.")
        return
    threading.Thread(target=server_task, daemon=True).start()
    agregar_log("Servidor iniciado.")

# Botón para detener el servidor
def detener_servidor():
    global server_running, server_socket
    if not server_running:
        agregar_log("El servidor no está en ejecución.")
        return
    server_running = False
    if server_socket:
        server_socket.close()
    agregar_log("Servidor detenido manualmente.")

# Agregar botones de control
boton_iniciar = tk.Button(
    ventana_principal,
    text="Iniciar Servidor",
    command=iniciar_servidor,
    font=("Arial", 12),
    bg="#3498db",  # Azul brillante
    fg="#ffffff"  # Texto blanco
)
boton_iniciar.pack(pady=10)

boton_detener = tk.Button(
    ventana_principal,
    text="Detener Servidor",
    command=detener_servidor,
    font=("Arial", 12),
    bg="#3498db",  # Azul brillante
    fg="#ffffff"  # Texto blanco
)
boton_detener.pack(pady=5)

# Ejecutar la ventana principal
ventana_principal.mainloop()

```

Productor – Consumidor:

El problema de Productor-Consumidor es un patrón de diseño clásico en la programación concurrente. En este modelo, un proceso productor genera datos y los coloca en una bodega o buffer (un área de almacenamiento compartida). Un proceso consumidor retira los datos del buffer para procesarlos. El reto radica en coordinar los productores y consumidores de manera eficiente y evitar problemas como la sobrecarga del buffer o el acceso simultáneo no sincronizado.



```python
import tkinter as tk
from queue import Queue
import random
import time

def ejecutar():
    # Tamaño máximo del buffer compartido
    BUFFER_SIZE = 5
    buffer = Queue(BUFFER_SIZE)

    # Variable de control para iniciar/detener procesos
    global running
    running = False

    # Función del Productor
    def productor():
        if running:
            if not buffer.full():
                item = random.randint(1, 100)  # Generar un ítem aleatorio
                buffer.put(item)  # Colocar el ítem en la cola
                log_text.insert(tk.END, "🔹 ")
                log_text.insert(tk.END, "Productor", "bold")
                log_text.insert(tk.END, f" produjo: {item}\n")
                log_text.see(tk.END)  # Desplazar automáticamente el scroll
            root.after(1000, productor)  # Programar la próxima ejecución

    # Función del Consumidor
    def consumidor():
        if running:
            if not buffer.empty():
                item = buffer.get()  # Extraer el ítem de la cola
                log_text.insert(tk.END, "🔸 ")
                log_text.insert(tk.END, "Consumidor", "bold")
                log_text.insert(tk.END, f" consumió: {item}\n")
                log_text.see(tk.END)  # Desplazar automáticamente el scroll
                buffer.task_done()  # Marca el ítem como procesado
            root.after(1500, consumidor)  # Programar la próxima ejecución

    # Iniciar los procesos
    def iniciar():
        global running
        if not running:
            running = True
            status_label.config(text="Estado: Activo 🟢🔵", fg="white")
            log_text.insert(tk.END, "🔥 Procesos iniciados...\n")
            productor()
            consumidor()

    # Detener los procesos
    def detener():
        global running
        if running:
            running = False
            status_label.config(text="Estado: Inactivo 🔴", fg="White")
            log_text.insert(tk.END, "🚫 Procesos detenidos...\n")

    # Configurar la ventana de Tkinter
    root = tk.Tk()
    root.title("Tigres Productor-Consumidor")
    root.geometry("550x500")
    root.config(bg="#34495e")  # Fondo gris oscuro

    # Crear el marco principal
    main_frame = tk.Frame(root, bg="#34495e")
    main_frame.pack(pady=10, padx=10, fill=tk.BOTH, expand=True)

    # Log para mostrar la actividad
    log_frame = tk.LabelFrame(main_frame, text="Log de Actividad", font=("Arial", 12, "bold"), bg="#2c3e50", fg="#ffffff", bd=3, relief="ridge")
    log_frame.pack(pady=10, padx=10, fill=tk.BOTH, expand=True)

    log_text = tk.Text(log_frame, width=60, height=15, state=tk.NORMAL, bg="white", fg="black", font=("Consolas", 10), bd=2, relief="sunken")
    log_text.tag_config("bold", font=("Consolas", 10, "bold"))  # Configuración para negrita
    log_text.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

    # Barra de estado
    status_label = tk.Label(root, text="Estado: Inactivo 🔴", fg="red", bg="#34495e", font=("Arial", 12, "bold"), bd=1, relief="sunken", anchor="w")
    status_label.pack(pady=5, fill=tk.X, padx=10)

    # Botones para controlar los procesos
    button_frame = tk.Frame(root, bg="#34495e")
    button_frame.pack(pady=10)

    start_button = tk.Button(button_frame, text="Iniciar 🔹", command=iniciar, bg="#3498db", fg="white", font=("Arial", 12, "bold"), width=12, relief="raised")
    start_button.grid(row=0, column=0, padx=10)

    stop_button = tk.Button(button_frame, text="Detener 🔴", command=detener, bg="#e74c3c", fg="white", font=("Arial", 12, "bold"), width=12, relief="raised")
    stop_button.grid(row=0, column=1, padx=10)

    # Ejecutar la aplicación
    root.mainloop()
```

Reactor – Proactor:

Los patrones Reactor y Proactor son utilizados en el diseño de sistemas asíncronos y de entrada/salida (I/O).

Reactor: Este patrón permite que un sistema maneje múltiples eventos de I/O de manera asíncrona. El reactor espera a que los eventos ocurran y luego distribuye el control a los manejadores de eventos correspondientes. El hilo principal sigue ejecutándose mientras espera estos eventos.

Proactor: Similar al reactor, pero en lugar de esperar a los eventos y luego gestionar su manejo, el Proactor delega la tarea a un manejador antes de que ocurra el evento, lo que implica un enfoque más proactivo.

```python
import asyncio

# Función que simula un evento asincrónico (Reactor Pattern)
async def handle_event(event_id):
    print(f"Handling event {event_id}")
    await asyncio.sleep(2)  # Simulando una tarea que toma tiempo
    print(f"Event {event_id} handled")

# Función Reactor que espera y maneja eventos
async def reactor():
    print("Reactor pattern started, waiting for events...")

    # Simulamos que ocurren varios eventos
    await asyncio.gather(
        handle_event(1),
        handle_event(2),
        handle_event(3)
    )
    print("Reactor pattern finished.")

# Función que simula un evento de I/O asincrónico (Proactor Pattern)
async def read_data():
    print("Reading data asynchronously...")
    await asyncio.sleep(3)  # Simulando una operación de I/O que toma tiempo
    print("Data read complete")

# Función Proactor que delega la ejecución de la tarea
async def proactor():
    print("Proactor pattern started, delegating tasks...")

    # Delegamos la operación de lectura de datos
    await read_data()

    print("Proactor pattern finished.")

# Función principal que maneja ambos patrones
async def main():
    print("Starting Reactor and Proactor patterns...\n")

    # Ejecutamos el Reactor Pattern
    await reactor()

    # Ejecutamos el Proactor Pattern
    await proactor()

    print("\nBoth Reactor and Proactor patterns have completed.")

# Ejecutar el programa
asyncio.run(main())
```

Modelo de Actores:

El modelo de actores es un modelo de programación en el que los actores son unidades independientes que pueden recibir mensajes, procesarlos y enviar mensajes a otros actores. Cada actor tiene su propio estado y la comunicación entre actores es asincrónica. Este modelo se utiliza en sistemas altamente concurrentes, como el modelo de actores de Erlang o Akka.

```python
import socket
import threading
import time
import random
from threading import Semaphore
import tkinter as tk
from tkinter import messagebox

# Configuración de los actores
HOST = '127.0.0.1'  # Dirección de loopback (localhost)
PORT_ACTOR1 = 5001  # Puerto para Actor1
PORT_ACTOR2 = 5002  # Puerto para Actor2

def ejecutar():
    # Contador de mensajes
    MAX_MESSAGES = 100
    message_counter = 0

    # Semáforos para controlar el acceso a los recursos compartidos
    semaphore_actor1 = Semaphore(1)
    semaphore_actor2 = Semaphore(1)

    # Estado de los hilos
    running = True

    # Función para escuchar mensajes de un puerto especifico
    def listen_actor(port, actor_name, semaphore, log_widget):
        nonlocal message_counter, running
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
            server_socket.bind((HOST, port))
            server_socket.listen()
            agregar_log(log_widget, f"{actor_name} escuchando en el puerto {port}")

            while message_counter < MAX_MESSAGES and running:
                conn, addr = server_socket.accept()
                with conn:
                    data = conn.recv(1024).decode('utf-8')
                    if data:
                        with semaphore:
                            message_counter += 1
                            agregar_log(log_widget, f"{actor_name} recibió: {data}")
                            agregar_log(log_widget, f"{actor_name} está ahora en estado: Ocupado")
                            time.sleep(1)  # Simulación de procesamiento

    # Función para enviar mensajes periódicamente a otro puerto
    def send_actor(target_port, actor_name, semaphore, log_widget):
        nonlocal message_counter, running
        while message_counter < MAX_MESSAGES and running:
            with semaphore:
                with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
                    try:
                        client_socket.connect((HOST, target_port))
                        message = f"Hola desde {actor_name}"
                        client_socket.sendall(message.encode('utf-8'))
                        agregar_log(log_widget, f"{actor_name} envió: {message}")
                        time.sleep(random.uniform(0.5, 1.5))  # Espera aleatoria
                    except ConnectionRefusedError:
                        agregar_log(log_widget, f"{actor_name} no pudo conectar a {target_port}")
                        time.sleep(1)

    # Función para agregar mensajes al log en la interfaz
    def agregar_log(widget, mensaje):
        widget.config(state="normal")
        widget.insert(tk.END, f"{mensaje}\n")
        widget.config(state="disabled")
        widget.see(tk.END)

    # Función para iniciar los actores
    def iniciar_actores(log_widget):
        nonlocal running, message_counter
        running = True
        message_counter = 0

        actor1_listen_thread = threading.Thread(target=listen_actor, args=(PORT_ACTOR1, "Actor1", semaphore_actor1, log_widget))
        actor2_listen_thread = threading.Thread(target=listen_actor, args=(PORT_ACTOR2, "Actor2", semaphore_actor2, log_widget))
        actor1_send_thread = threading.Thread(target=send_actor, args=(PORT_ACTOR2, "Actor1", semaphore_actor1, log_widget))
        actor2_send_thread = threading.Thread(target=send_actor, args=(PORT_ACTOR1, "Actor2", semaphore_actor2, log_widget))
```

```python

    # Iniciar hilos
    actor1_listen_thread.start()
    actor2_listen_thread.start()
    actor1_send_thread.start()
    actor2_send_thread.start()

# Función para detener los actores
def detener_actores():
    nonlocal running
    running = False
    # Eliminar la ventana emergente
    # messagebox.showinfo("Estado", "Actores detenidos.")

# Configuración de la ventana principal
ventana_principal = tk.Tk()
ventana_principal.title("Interacción entre Actores")
ventana_principal.geometry("600x500")
ventana_principal.configure(bg="#34495e")  # Fondo gris oscuro

# Título
etiqueta_titulo = tk.Label(
    ventana_principal,
    text="Actores en Interacción - ¡Conectados Siempre!",
    font=("Arial", 14),
    bg="#34495e",  # Fondo gris oscuro
    fg="#ffffff"  # Texto blanco
)
etiqueta_titulo.pack(pady=10)

# Cuadro de texto para mostrar el log
cuadro_log = tk.Text(
    ventana_principal,
    height=20,
    width=70,
    state="disabled",
    font=("Arial", 10),
    bg="#ffffff",  # Fondo blanco para la terminal
    fg="#000000"  # Texto negro
)
cuadro_log.pack(pady=10)

# Botón para iniciar actores
boton_iniciar = tk.Button(
    ventana_principal,
    text="Iniciar Actores",
    command=lambda: iniciar_actores(cuadro_log),
    font=("Arial", 12),
    bg="#3498db",  # Fondo azul brillante
    fg="#ffffff"  # Texto blanco
)
boton_iniciar.pack(pady=5)

# Botón para detener actores
boton_detener = tk.Button(
    ventana_principal,
    text="Detener Actores",
    command=detener_actores,
    font=("Arial", 12),
    bg="#3498db",  # Fondo azul brillante
    fg="#ffffff"  # Texto blanco
)
boton_detener.pack(pady=5)

# Ejecutar la ventana principal
ventana_principal.mainloop()

```

Tkinter:

Tkinter es una biblioteca estándar de Python para crear interfaces gráficas de usuario (GUI). Proporciona herramientas para crear ventanas, botones, cuadros de texto, menús y otros elementos gráficos en una aplicación. Tkinter está basado en la biblioteca Tk, que es un conjunto de herramientas para la creación de interfaces gráficas que funciona de manera multiplataforma.

Explicación de las Funciones Principales y Códigos

Este código es un programa que tiene como objetivo enseñar conceptos de programación concurrente utilizando una interfaz gráfica. Básicamente, la aplicación permite al usuario ejecutar diferentes ejemplos de programación con hilos, sockets, semaforos y patrones de diseño (como productor-consumidor, actores, etc.) y leer documentación relacionada con estos temas. Está hecho en Python y usa la librería Tkinter para la interfaz gráfica y otras librerías como PIL (para mostrar imágenes) y fitz (para mostrar archivos PDF).

Interfaz Gráfica:

El programa tiene una ventana principal, en la que se usan botones para interactuar con el usuario. Al hacer clic en cada botón, el usuario puede acceder a diferentes secciones como "Hilos", "Sockets", "Semáforos", "Patrones" y "Documentación".

Tkinter se usa para crear la interfaz. Se define un fondo de pantalla, un menú en la parte superior con botones que hacen acciones, y ventanas emergentes que muestran el contenido.

Menú Principal:

En la ventana principal hay botones que, al hacer clic, nos muestran un submenú con más opciones. Por ejemplo, si hacemos clic en el botón de "Hilos", veremos opciones como "Hilos con argumentos", "Hilos sincronizados", etc. Cada opción ejecuta una función relacionada con ese tema.

Si seleccionamos una opción, el programa ejecutará una función definida previamente en módulos de Python que importamos (como hilos_hilos, hilos_con_argumentos, etc.).

Mostrar Documentación:

El programa también tiene botones que permiten leer documentos en PDF relacionados con los temas. Por ejemplo, si seleccionas "Apunte De Hilos", el programa abrirá un archivo PDF con información sobre hilos en programación concurrente.

Los archivos PDF se muestran como imágenes en una ventana emergente gracias a la librería fitz y PIL. Esto es porque la librería fitz convierte las páginas del PDF en imágenes, y luego las mostramos en la interfaz.

Funciones de Hilos, Sockets, Semáforos, y Patrones:

En la sección de hilos, por ejemplo, se pueden ejecutar ejemplos que muestran cómo crear y manejar hilos en Python (como pasarles argumentos o sincronizarlos).

En la parte de sockets, se puede aprender sobre cómo hacer que dos programas se comuniquen a través de redes (por ejemplo, usando TCP o UDP).

Semáforos son útiles para gestionar el acceso a recursos compartidos entre varios hilos, evitando problemas como la condición de carrera. El programa también permite ver ejemplos de esto, como el clásico "barbero dormilón".

Los patrones de diseño son formas de estructurar la programación para hacerla más eficiente y comprensible. En este caso, uno de los patrones que se puede explorar es el Productor/Consumidor y el Reactor y Proactor.

Documentación y Acerca de:

Además de los ejemplos de código, el programa incluye una sección de documentación que tiene PDFs de teoría sobre cada uno de los temas que el programa cubre (hilos, sockets, semáforos, etc.).

También hay un botón de "Acerca de" donde se explica quiénes son los autores del proyecto y se dan detalles del equipo que lo hizo.

Cerrar la Aplicación:

Finalmente, el programa tiene una opción para cerrar la aplicación de manera ordenada, cerrando también las ventanas adicionales que se hayan abierto durante la ejecución.

Herramientas de Programación Utilizadas

Librerías Integradas:

- tkinter: Construcción de la GUI.
- subprocess: Ejecución de scripts externos.
- os: Operaciones del sistema.
- Librerías Externas:
- Pillow: Manipulación de imágenes.
- fitz (PyMuPDF): Renderización de PDFs.

Estructura Modular:

Los scripts como hilos_hilos.py son módulos independientes, lo que facilita su mantenimiento.

Conclusión

El programa es una herramienta educativa práctica que combina una GUI funcional con conceptos avanzados de programación concurrente. Su diseño modular asegura facilidad para futuras extensiones, y su documentación detallada respalda el aprendizaje y trabajo en equipo.

Bibliografía

- Ben-Ari, M. (2006). Principles of Concurrent and Distributed Programming (2nd ed.). Pearson Education.
- Rossainz López, M. (s.f.). Programación Concurrente y Paralela. Universidad Autónoma de Puebla.
- Andrews, G. R. (1991). Concurrent Programming: Principles and Practice. Addison-Wesley.
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.).

Glosario de Términos Técnicos

- Hilos: Unidades básicas de procesamiento concurrente.
- Semáforos: Herramientas de sincronización para recursos compartidos.
- Sockets: Interfaces para comunicación cliente-servidor.
- Patrones de Diseño: Soluciones genéricas para problemas recurrentes en el desarrollo de software.
- PyMuPDF: Biblioteca para manipulación de PDFs.
- GUI: Interfaz gráfica para interacción usuario-programa.
- Futuro - Promesa: Mecanismos para manejar operaciones asincrónicas que devuelven un valor en el futuro.
- Productor - Consumidor: Patrón de diseño para coordinar procesos productores y consumidores con recursos compartidos.
- Reactor - Proactor: Patrones de diseño para manejar eventos asincrónicos en sistemas de I/O.
- Modelo de Actores: Modelo de programación en el que los actores son unidades independientes que interactúan mediante mensajes.
- Tkinter: Biblioteca de Python para construir interfaces gráficas de usuario (GUI).